20MCA 132 Object Oriented Programming Lab

CO4: Implement multithreading, files, java.util package and Collection framework

File handling in java

The File class from the java.io package, allows us to work with files. To use the File class, create an object of the class, and specify the filename or directory name. First of all, we should create the File class object by passing the filename or directory name to it. A File object is created by passing in a string that represents the name of a file, a String, or another File object. For example,
File myObj = new File("filename.txt"); // Specify the filename
File f=new File("s2MCA"); // Specify the directory name

Return type	Method	Description
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.

long	length()	Returns the size of the file in bytes
String[]	list()	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
boolean	mkdir()	It creates the directory named by this abstract pathname.
string	getName()	It returns the name of the file or directory denoted by this abstract pathname.
string	getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory
boolean	delete()	Deletes the file or directory denoted by this abstract pathname.

Example 1:

```
import java.io.*;
public class FileDemo {
   public static void main(String[] args) {

      try {
         File file = new File("javaFile123.txt");
         if (file.createNewFile()) {
            System.out.println("New File is created!");
         } else {
            System.out.println("File already exists.");
         }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }       } }
```

Example 2

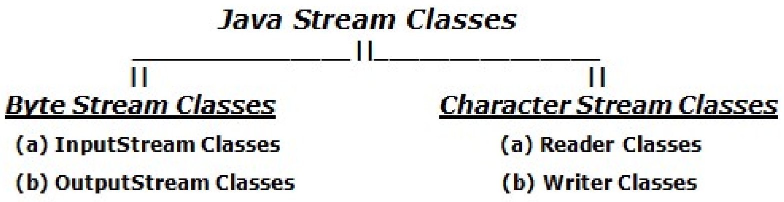
```
import java.io.File;
 // Displaying file property
class fileProperty {
    public static void main(String[] args)    {
         // accept file name or directory name through command line args
        String fname = args[0];
         // pass the filename or directory name to File object
        File f = new File(fname);
         // apply File class methods on File object
        System.out.println("File name :" + f.getName());
        System.out.println("Path: " + f.getPath());
        System.out.println("Absolute path:“ + f.getAbsolutePath());
        System.out.println("Parent:" + f.getParent());
        System.out.println("Exists :" + f.exists());
         if (f.exists()) {
            System.out.println("Is writable:“ + f.canWrite());
            System.out.println("Is readable" + f.canRead());
            System.out.println("Is a directory:“ + f.isDirectory());
            System.out.println("File Size in bytes "+ f.length());
        }
 }
}
```

Stream classes in Java

In Java, a **stream** is a **path** along which the **data flows**. Every stream has a **source** and a **destination**. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination. InputStream and OutputStream are the basic stream classes in Java. Java encapsulates Stream under **java.io** package. Java defines two types of streams.

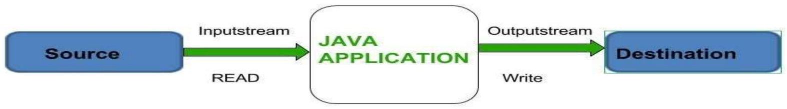
- . The java.io package contains a large number of stream classes that provide capabilities for processing all types of data.

- These classes may be categorized into two groups based on the data type on which they operate.
    - **Byte stream classes -** These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc.
    - **Character stream classes -** These handle data in 16 bit Unicode. Using these you can read and write text data only.

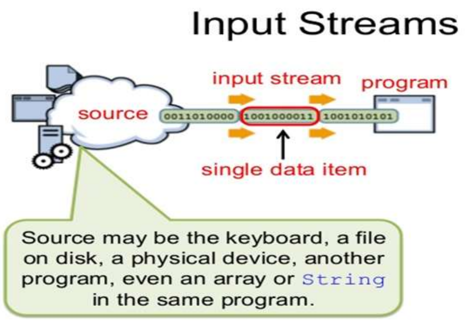


Byte Stream Classes

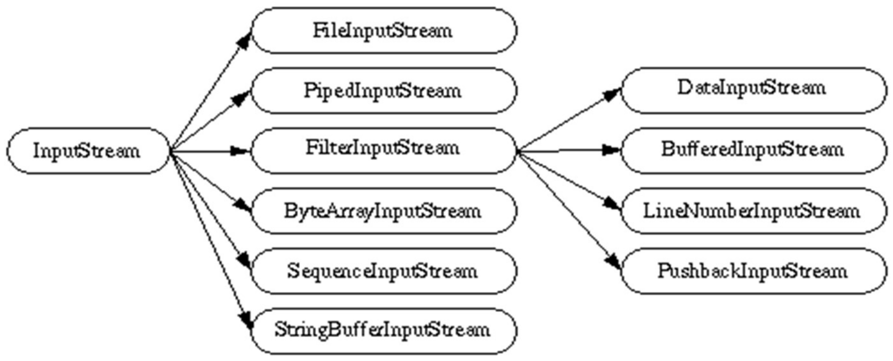
- Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes.
- Java provides two kinds of byte stream classes:
    - **InputStream classes**
    - **OutputStream classes**.



- **Input Stream Classes**
  - Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.
  - Input stream classes that are used to read bytes include a super class known as **Inputstream** and a number of subclasses for supporting various input-related functions.
  - The super class InputStream is an **abstract** class, and, therefore, we cannot create instances of this class. Rather, we must use the subclasses that inherit from this class.


Input Streams
input stream
program
source
0011010000
1001000011
1001010101
single data item
Source may be the keyboard, a file on disk, a physical device, another program, even an array or String in the same program.

- The InputStream class of the java.io package is an abstract superclass that represents an input stream of bytes.
- Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input.
- ***Hierarchy of InputStream class***



- **Methods of InputStream**
    - **InputStream classes performs the following operations.**
        - **Reading Bytes**
        - **Finding the number of bytes in the stream.**
        - **Marking the position in stream.**
        - **Skipping**
        - **Closing the stream**

    - The InputStream class provides different methods that are implemented by its subclasses.
    - Here are some of the commonly used methods:

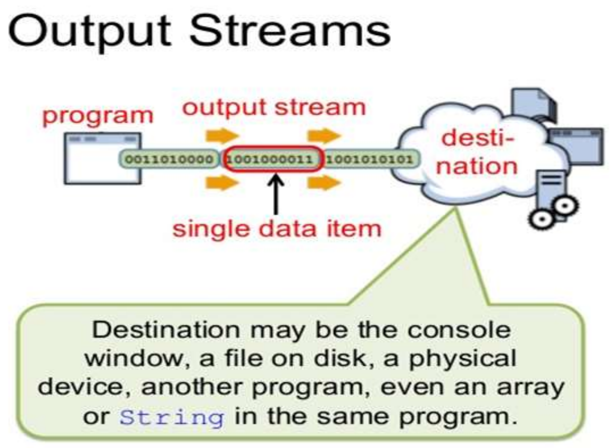
- **int available( )**
    - This method returns an estimate of the number of bytes that can be read from this input stream.
- **int read(byte b[ ])**
    - This method reads some number of bytes from the input stream and stores them into the buffer array b.
    - -1 will be returned if it's end of file.
- **int read(byte b[ ], int n, int m)**
    - This method reads up to m bytes of data from the input stream into an array of bytes.

- This method also reads bytes into a byte array, but starts at nth bytes into the array, and reads a maximum of m bytes into the array from that position

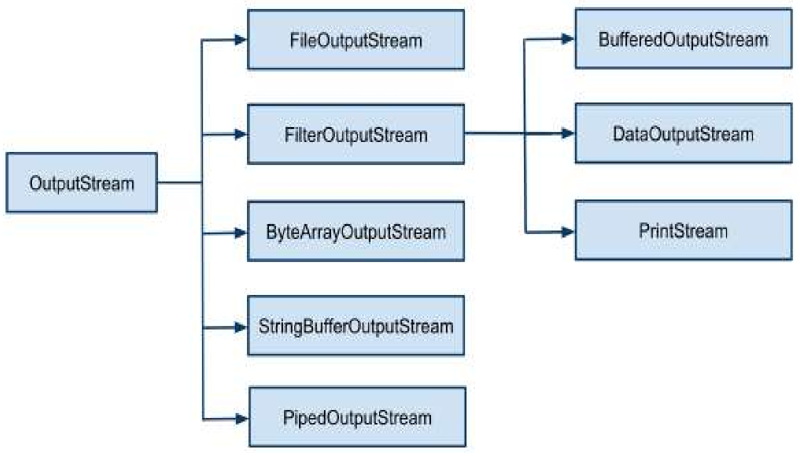
- **void close( )**
  - This method closes this input stream and releases any system resources associated with the stream.
- **void mark(int readlimit)**
  - This method marks the current position in this input stream.
  - marks the position in the input stream up to which data has been read.
- **void reset( )**
  - This method repositions this stream to the position at the time the mark method was last called on this input stream.
- **long skip(long n)**

  This method skips over and discards n bytes of data from this input stream.

- **<u>OutputStream</u>**
  - Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.
  - Output stream classes are derived from the base class **Outputstream** like InputStream, the OutputStream is an **abstract** class and therefore we cannot instantiate it.
  - The several subclasses of the OutputStream can be used for performing the output operations.


Output Streams
program
output stream
0011010000
1001000011
1001010101
desti-
nation
single data item
Destination may be the console window, a file on disk, a physical device, another program, even an array or String in the same program.

Hierarchy of OutputStream class



Methods in OutputStream classes

void write(byte b[])

- Writes an entire length of an array of bytes to the output stream.

void write(byte b[], int n, int len)

- Writes len bytes from the specified byte array b, starting at n off to this output stream.

void flush( )

- Flushes this output stream.
- Forces to write all data present in output stream to the destination.

void close( )

- Closes this output stream and releases any system resources associated with this stream.

FileInputStream

- The FileInputStream class of the java.io package can be used to read data from files.
- Common constructor is
    - **FileInputStream(String filepath)**
- ✓ This creates a FileInputStream by opening a connection to an actual file.
- ✓ the file named by the path name filepath in the file system
- ✓ FileInputStream **example 1: read single character**

```
import java.io.FileInputStream;
 class fileinputdemo
{
        public static void main(String args[ ])    {
                try     {
               FileInputStream fin=new FileInputStream("test1.txt");
                int i=fin.read();
                 System.out.print((char)i);
                 fin.close();
                }
                catch(Exception e)
                {System.out.println(e);}
          }
}
```

- Before running the code, a text file named as test1.txt is required to be created.
- In this file, we are having following content:
    - Welcome to java
- After executing the above program, we will get a single character from the file which is 87 (in byte form).
- To see the text, we need to convert it into character.

Output:

```
W
```

FileInputStream Example 2: read all characters

```
import java.io.FileInputStream;
 class  fileinputdemo2
{
        public static void main(String args[ ])
        {
        try
                {
                   FileInputStream fin=new FileInputStream (“test1.txt");
                   int i=0;
                   while((i=fin.read())!=-1)
                    {
                       System.out.print((char)i);
                    }
                    fin.close();
                 }
                catch(Exception e)
                       {System.out.println(e);}
      }      }
```

FileOutputStream Class

- The FileOutputStream class of the java.io package can be used to write data in bytes to the files.
- Common constructor are
    - **FileOutputStream(String filepath)**
        - Creates a file output stream to write to the file with the specified name.
        - The specified name of the file with path name of the file is specified in filepath.
    - **FileOutputStream(String filepath, boolean value)**

- Here, we have created an output stream that will be linked to the file specified by the filepath.
- Also, value is an optional boolean parameter.
- If it is set to true, the new data will be appended to the end of the existing data in the file. Otherwise, the new data overwrites the existing data in the file.

FileOutputStream- E**xample 1: write a single character**

```
import java.io.FileOutputStream;
 class FileOutputdemo1
 {
        public static void main(String args[ ])    {
          try{
                FileOutputStream fout=new FileOutputStream("test2.txt");
                fout.write(65);
                fout.close( );
                System.out.println("success...");
               }
               catch(Exception e)
               {
                       System.out.println(e);
               }
         }   }
```

The content of a text file test2.txt is set with the data A.

- **FileOutputStream- Example 2: write string**

```
import java.io.FileOutputStream;
 class FileOutputdemo2
 {
         public static void main(String args[ ])   {
         try{
                 FileOutputStream fout=new FileOutputStream("test2.txt");
```

```
                    String s="Welcome to java.";
                    byte b[ ]=s.getBytes( );//converting string into byte array
                    fout.write(b);
                    fout.close( );
                     System.out.println("success...");
                   }
                  catch(Exception e)
                  {
                  System.out.println(e);
                  }
                }
}
```

Example program with methods in FileInputStream class

```
import java.io.*;
class fileinputdemo3
{
    public static void main(String args[ ])      {
          FileInputputStream f=new FileInputStream("fileinputdemo3.java");
         int size=f.available( );
          System.out.println("Total Size="+size);
         int n=size/40;
          System.out.println("Now the size is="+n);
         for(int i=0;i<n;i++)
         {
                  System.out.print((char)f.read( ));
         }
           System.out.println( )
         int a=f.available( );
          System.out.println("Still Available="+a);
         int y=f.skip(a/2);
          System.out.println("Skipped="+y);
         System.out.println("Now the new file is");
         for(int i=0;i<y;i++)
         {
                  System.out.print((char)f.read( ));
```

```
        }
        f.close( );
    }
  }
```

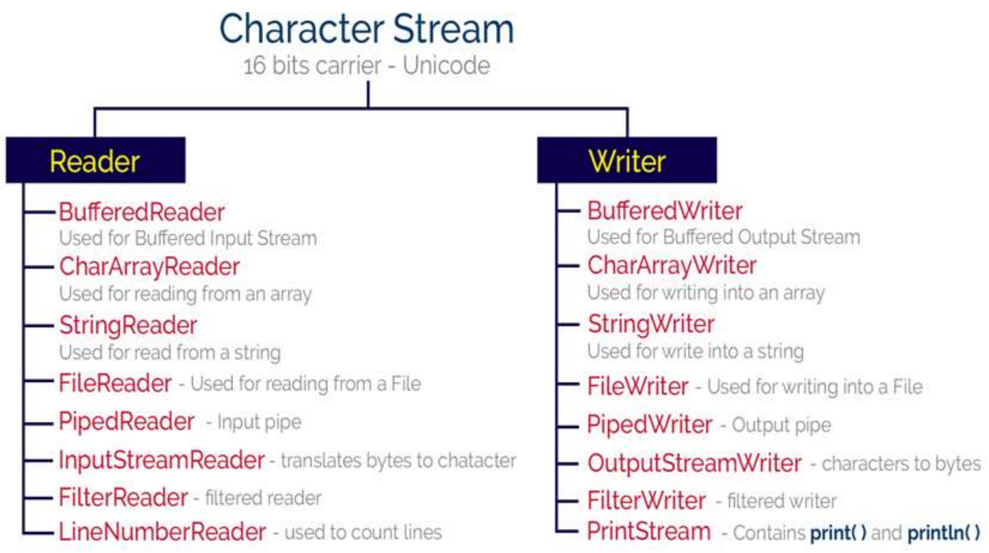
Output

```
Total Size=669
Now the size is=16
import java.io.F
Still Available=653
Skipped=326
Now the new file is
      {      System.out.print((char)f.read());
      }
       System.out.println();
      int a=f.available();
      System.out.println("Still Available="+a);
      long y=f.skip(a/2);
      System.out.println("Skipped="+y);
      System.out.println("Now the new file is");
      for(int i=0;i<y;i++)
      {
      System.out.print((char)f.read());
      }
      f.close();
    }
```

Character Stream classes

- In java, when the IO stream manages 16-bit Unicode characters, it is called a character stream.
- The java character stream is defined by two abstract classes,
    - **Reader** and
    - **Writer**.

- The Reader class used for character stream based input operations, and the Writer class used for character stream based output operations.
- The Reader and Writer classes have several concrete classes to perform various IO operations based on the character stream


Character Stream
16 bits carrier - Unicode
Reader
BufferedReader
Used for Buffered Input Stream
CharArrayReader
Used for reading from an array
StringReader
Used for read from a string
FileReader - Used for reading from a File
PipedReader - Input pipe
InputStreamReader - translates bytes to chatacter
FilterReader - filtered reader
LineNumberReader - used to count lines
Writer
BufferedWriter
Used for Buffered Output Stream
CharArrayWriter
Used for writing into an array
StringWriter
Used for write into a string
FileWriter - Used for writing into a File
PipedWriter - Output pipe
OutputStreamWriter - characters to bytes
FilterWriter - filtered writer
PrintStream - Contains print( ) and println( )

Multithreading

Threads

- A thread represents a separate path of execution of a group of statements.
- The way statements are executed are classified into two types:
  1) Single Tasking
  2) Multi Tasking

<u>Multi Tasking</u>

Suppose the processor starts at the first job. Then it will spend exactly ¼ milliseconds for the first job. If the processor is not able to complete the job within this time, it will store the intermediate results in a temporary memory and it goes to the second task. It spends exactly ¼ milliseconds for the second task. After that it proceeds in the same manner or 3rd and 4th tasks.

After executing the fourth task for ¼ milliseconds, it will come back to the first task, in a circular manner. This is called round robin method.

The processor after returning to first task resumes it from the point where it has left the first task earlier and executes it for ¼ milliseconds and proceeds in the round–robin manner. Here the processor is executing one job for ¼ milliseconds and keeping it waiting for another ¾ millisecond, while it is going round executing the other tasks. After ¾ milliseconds it will again execute first job for ¼ milliseconds. We will feel that the processor spends time executing for one job only. So in multitasking, there will be a single processor performing multiple tasks.

It is achieved by providing each process a time-slice and executing them in round robin manner ,so that we will get a feel that the processor is spending time for your task only. The main advantage of multi taking is to use processor time in a better way. Here most of the processor time is engaged and it is not sitting idle. In this way we can complete several tasks at a time, and thus achieve good performance.

- Multi tasking are of two types:
  - a) Process-based multi tasking
  - b) Thread based multi tasking

Process, is a program that is currently executing. Thus, process-based multitasking is the feature that allows our computer to run two or more programs concurrently. For example, process based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Inter-process communication is expensive and limited. Context switching from one process to another is also costly.

Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Multithreaded multitasking is controlled by java. Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Multithreading

Multithreading is a conceptual programming paradigm where a program is divided into two or more subprograms, which can be implemented at the same time in parallel. A thread is similar to a program that has a single flow of control. It has a beginning, a body and an end, and commands executes sequentially. In fact all main program in our earlier examples can be called single-threaded programs.

- A single threaded pgm. Is shown below,

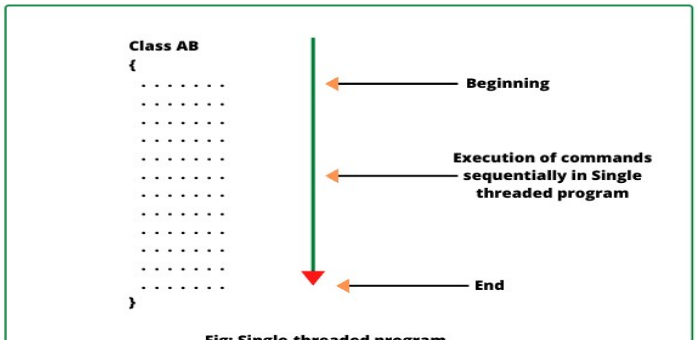
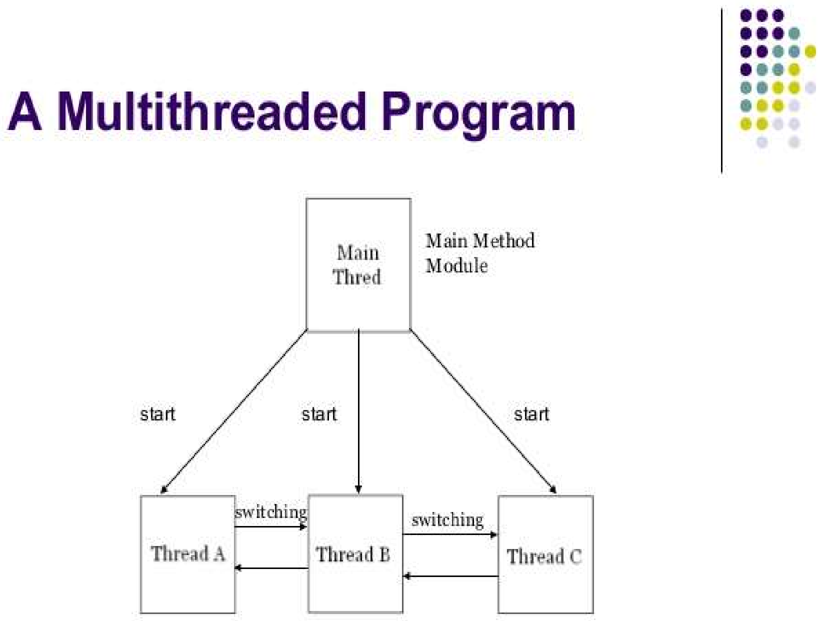

Class AB
{
}
Beginning
Execution of commands sequentially in Single threaded program
End

Fig: Single-threaded program

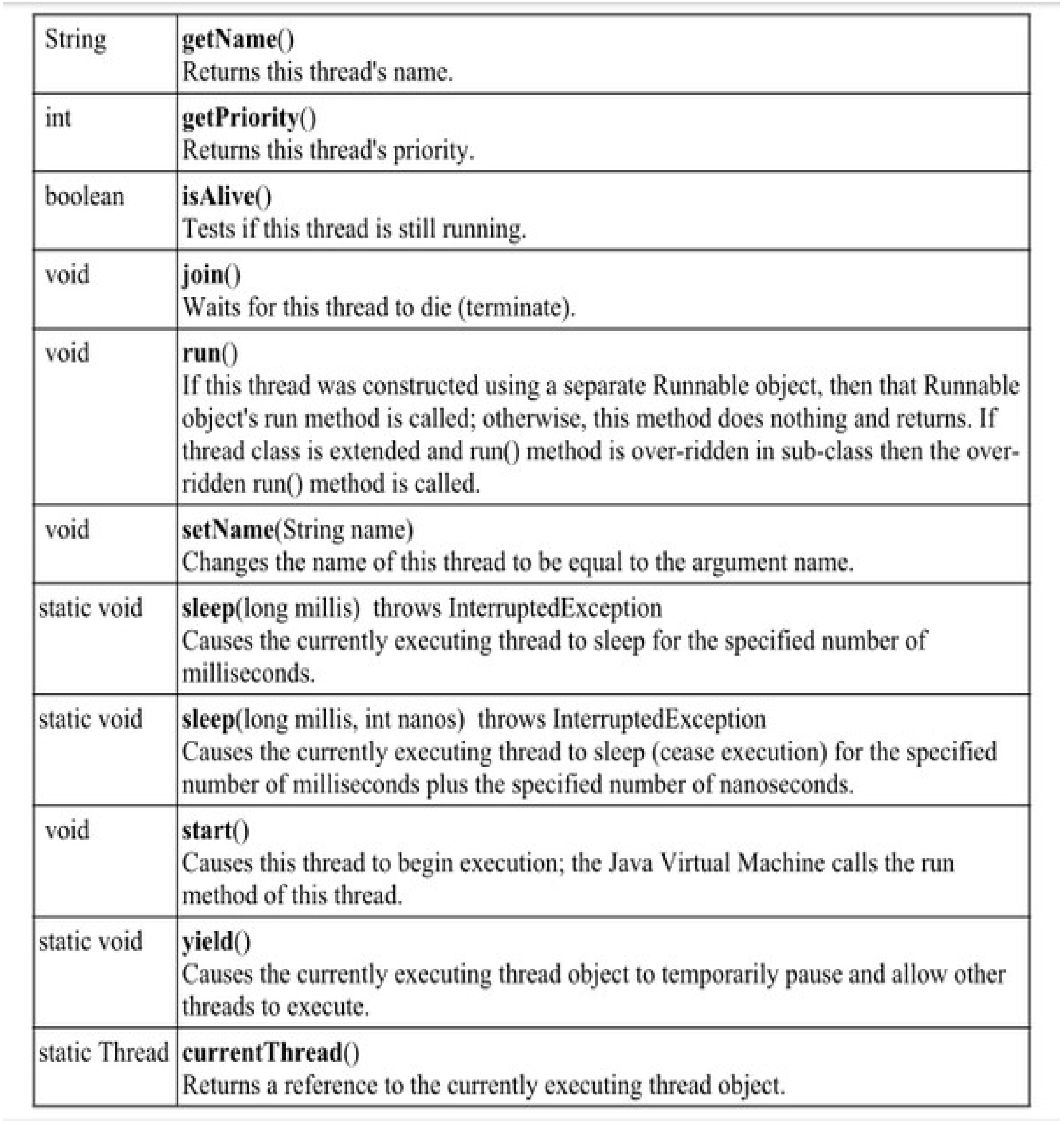
- A unique property of Java is its support of multithreading.
- That is, Java enables us to use multiple flows of control in developing programs.
- Each flow of control may be thought of as a separate tiny pgm. or module known as a thread that runs in parallel.
- A program that contains multiple flows of control is known as multithreaded pgm.

The following fig. have 4 threads, one main and 3 others,



- The main thread is actually the main method module, which is designed to create and start the other 3 threads namely A, B and C.
- Java‘s multithreading system is built upon the
  - Thread class and its methods,

    or
  - The interface, Runnable.
- To create a new thread, our program will either extend Thread class or implement the Runnable interface.
- The Thread class defines several methods that help to manage thread:



String	**getName**() Returns this thread's name.
int	**getPriority**() Returns this thread's priority.
boolean	**isAlive**() Tests if this thread is still running.
void	**join**() Waits for this thread to die (terminate).
void	**run**() If this thread was constructed using a separate Runnable object, then that Runnable object's run method is called; otherwise, this method does nothing and returns. If thread class is extended and run() method is over-ridden in sub-class then the over-ridden run() method is called.
void	**setName**(String name) Changes the name of this thread to be equal to the argument name.
static void	**sleep**(long millis) throws InterruptedException Causes the currently executing thread to sleep for the specified number of milliseconds.
static void	**sleep**(long millis, int nanos) throws InterruptedException Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds.
void	**start**() Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
static void	**yield**() Causes the currently executing thread object to temporarily pause and allow other threads to execute.
static Thread	**currentThread**() Returns a reference to the currently executing thread object.

The main thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when our program begins. The main thread is created automatically when our program is started. It can be controlled through a thread object by calling the method **currentThread( )**.

- **Example program,**

```
class mainthread
 {
          public static void main(String args[])
         {
                  Thread t = Thread.currentThread();
                  System.out.println("Current Thread="+t);
                  t.setName("My Thread");
                  System.out.println("After name change="+t);
                  try{
                           for(int i=5;i>0;i--)
                            {
                           System.out.println(i);
                           Thread.sleep(1000);
                            }
                    }
                  catch(Exception e)
                  {
                       System.out.println(" Main thread interrupted");
                  }
         }
}
```

<u>Output</u>

```
Current Thread=Thread[main,5,main]
After name change=Thread[My Thread,5,main]
5
4
3
2
1
```

- A reference to the current thread is obtained by calling the method currentThread().
- The method setName() is used to change the internal name of the thread.
- sleep() method is used to pausing one second between each line.

- The sleep() method Thread class might throw an InterruptedException.
- This would happens if some other thread wanted to interrupt this sleep one.
- The object 't' displays, the name of the thread, its priority and the name of its group.
- By default the name of the main thread is main.
- Its priority is 5 by default and main is also the name of the group of the thread to which this thread belongs.

Creating Thread

- A new thread can be created in 2 ways:
  - By extending Thread class
  - By implementing Runnable interface

1. <u>By extending Thread class</u>

- Thread class is in java.lang package
- Steps to create a new thread are:
  a) Declare the class as extending the Thread class
  b) Override the run() method
  c) Create an instance of this new class
  d) Invoke the start() method on the instance

a) Declare the class as extending the Thread class

- An instance of the java.lang.Thread class is associated with each thread running in the JVM.
- These Thread objects serve as the interface for interacting with the underlying operating system thread.
- It is possible to extend the Thread class in our class.
- Then creating objects of class acts as threads.
- Through the methods in the Thread class, new threads also can be started, stopped, interrupted, named and prioritized regarding their current state.
- The syntax for extending Thread class is,

```
class mythread extends Thread
{
        --------------
}
```

b) Override the run() method

- After extending Thread, the next step is to override the run() method.
- The syntax is,

```
public void run()
{
                ---------
}
```

- When a new thread is started, the entry point into the program is the run() method.
- The first statement in run() will be the first statement executed by the new thread.
- Every statement that the thread will execute is included in the run() method or is in other methods invoked directly or indirectly by run().
- The new thread is considered to be alive from just before run() is called until just after run() returns, at which time the thread dies.
- After a thread has died, it cannot be restarted.

c) Create an instance of this new class

- Once the class which extends the thread class is defined we need to create an instance of the newly created class in order to execute that thread.
- For eg.

```
        mythread a= new mythread();
```

d) Invoke the start() method

- After the new thread is created, it will not start running until we call its start( ) method, which is declared within Thread.
- The start() method causes and not actually starts execution.
- It schedules the thread and when CPU scheduler picks this thread for execution then JVM calls the run() method to actually start execution.
- Thread can call start() method only once in a program.
- A thread will throw 'IllegalStateException' if we try to call start method on already started thread instance.
- The syntax for starting the thread for execution is,

```
                a.start();
```

- This statement calls the start() method and move the thread into the runnable state.

Example program

```
class mythread extends Thread
{
        mythread()
        {
          super("Demo Thread");
          System.out.println("Child Thread="+this);
          start();
        }
        public void run()
        {
          try{
                for(int i=5;i>0;i--)
                {
                  System.out.println("Child Thread:"+i);
                  Thread.sleep(500);
                }
            }
        catch(Exception e)
          {     System.out.println("Chaild Thread Interrupted");}
                System.out.println("Exiting Child Thread");
        }
}
class extendThread
{
   public static void main(String arg[])
   {
        mythread obj=new mythread();
        try{
                for(int j=5;j>0;j--)
                {
                  System.out.println("Main Thread:"+j);
                  Thread.sleep(1000);
                }
            }
          catch(Exception e)
          {
                System.out.println("Main Thread Interrupted");
            }
            System.out.println("Exiting Main Thread");
        }
}
```

Output

```
Child Thread=Thread[Demo Thread,5,main]
Main Thread:5
Child Thread:5
Child Thread:4
Main Thread:4
Child Thread:3
Child Thread:2
Main Thread:3
Child Thread:1
Exiting Child Thread
Main Thread:2
Main Thread:1
Exiting Main Thread
```

- Here mythread class extends Thread class.
- extendThread class creates an object of the mythread class.
- The constructor of mythread class calls the super() method to sets the name of the child thread as "Demo Thread".
- The start() method calls the run() method for starting the execution of the child thread.
- The constructor then returns the control to main main method.
- When the main thread resumes, it enters its for loop.
- Both threads contains running, sharing the CPU until their loops finish.

2. By implementing Runnable interface

a) Declare a class as implementing the Runnable interface.

- For eg.

```
class x implements Runnable
{
```

```
                ------------
        }
```

b) Override the run() method
c) Create a Thread object with in the constructor of a class which implements Runnable interface.
d) Call the start() method to run the thread.

Example Program

```
class newThread implements Runnable
{
        Thread t;
        newThread()
        {
          t=new Thread(this,"Demo Thread");
          System.out.println("Child thread Name:"+t);
          t.start();
        }
        public void run()
        {
         try{
                for(int i=5;i>0;i--)
                {
                  System.out.println("Child thread:"+i);
                   Thread.sleep(500);
                }
            }
        catch(Exception e)
          {
                System.out.println("Child thread interrupted");
           }
           System.out.println("Exiting from Child thread");
        }
}
class threaddemo
{
        public static void main(String arg[])
        {
                newThread obj=new newThread();

                try
                {
                  for(int j=5;j>0;j--)
                  {
                        System.out.println("Main thread:"+j);
                        Thread.sleep(1000);
                   }
                }
                catch(Exception e)
                {
                System.out.println("Main thread interrupted");
                }
                System.out.println("Exiting from Main thread");
        }
}
```

Output

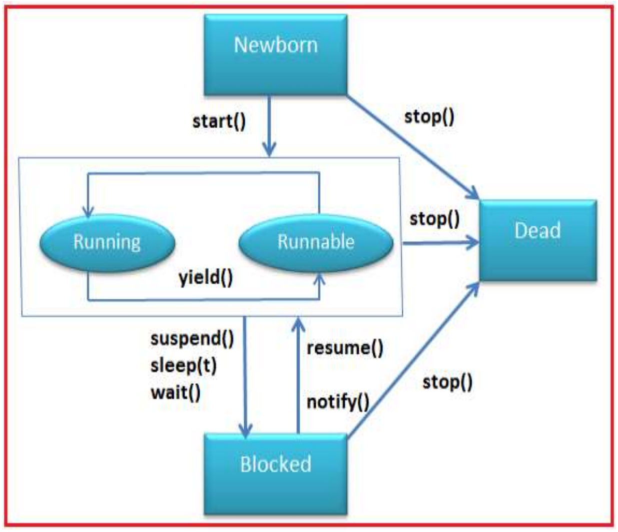
```
Child thread Name:Thread[Demo Thread,5,main]
Main thread:5
Child thread:5
Child thread:4
Child thread:3
Main thread:4
Child thread:2
Child thread:1
Main thread:3
Exiting from Child thread
Main thread:2
Main thread:1
Exiting from Main thread
```

Thread Life Cycle

- A thread goes through various stages in its life cycle.

- The life cycle of the thread in java is controlled by JVM.
- The java thread states are as follows:
  - **Newborn**
  - **Runnable**
  - **Running**
  - **Blocked (Non-Runnable)**
  - **Dead (Terminated)**

- The diagram shown below represent various states of a thread at any instant of time.



- **New**
  - A new thread begins its life cycle in the new state.
  - It remains in this state until the program starts the thread.
  - It is also referred to as a born thread.
  - In simple words, a thread has been created, but it has not yet been started.
  - A thread is started by calling its start() method.

- **Runnable**
  - The thread is in the runnable state after the invocation of start() method, but the thread scheduler has not selected it to be the running thread.
  - A thread starts life in the Ready-to-run state by calling the start method and wait for its turn.
  - The thread scheduler decides which thread runs and for how long.

- **Running**
  - When the thread starts executing, then the state is changed to a "running" state.
  - The scheduler selects one thread from the thread pool, and it starts executing in the application.

- **Dead**
  - This is the state when the thread is terminated.
  - The thread is in running state and as soon as it completed processing it is in "dead state".
  - Once a thread is in this state, the thread cannot even run again.

- **Blocked (Non-runnable state):**
  - This is the state when the thread is still alive but is currently not eligible to run.
  - A thread that is blocked waiting for a monitor lock is in this state.
  - A running thread can transit to one of the non-runnable states depending on the situation.
  - A thread remains in a non-runnable state until a special transition occurs.
  - A thread doesn't go directly to the running state from a non-runnable state but transits first to the Ready-to-run state.

Thread Priority

In a Multi threading environment, thread scheduler assigns processor to a thread based on priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by programmer explicitly. Default priority of a thread is 5 (NORM_PRIORITY). Accepted value of priority for a thread is in range of 1 to 10.

- There are 3 static variables defined in Thread class for priority.
  - **public static int MIN_PRIORITY:** This is minimum priority that a thread can have. Value for this is 1.
  - **public static int NORM_PRIORITY:** This is default priority of a thread if do not explicitly define it. Value for this is 5.
  - **public static int MAX_PRIORITY:** This is maximum priority of a thread. Value for this is 10.

Get and Set methods in Thread priority

1. public final int getPriority()

- ☐ In Java, getPriority() method is in java.lang.Thread package. it is used to get the priority of a thread.

2. public final void setPriority(int newPriority)

- ☐ In Java setPriority(int newPriority) method is in java.lang.Thread package.
- ☐ It is used to set the priority of a thread.
- ☐ The setPriority() method throws IllegalArgumentException if the value of new priority is above minimum and maximum limit.

Example 1

```
class ThreadDemo extends Thread
 {
   public void run()
   {
      System.out.println("Inside run method");
   }
 }
 class Treadpriority
{
   public static void main(String[] args)
   {
      ThreadDemo t1 = new ThreadDemo();
      ThreadDemo t2 = new ThreadDemo();
      ThreadDemo t3 = new ThreadDemo();

      // Default 5
      System.out.println("t1 thread priority : "+ t1.getPriority());

      // Default 5
      System.out.println("t2 thread priority : "+ t2.getPriority());
         // Default 5
      System.out.println("t3 thread priority : "+ t3.getPriority());
```

```
        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);
        // t3.setPriority(21); will throw IllegalArgumentException
        System.out.println("t1 thread priority :"+ t1.getPriority());
        System.out.println("t2 thread priority :"+ t2.getPriority());
        System.out.println("t3 thread priority :"+ t3.getPriority());
         // Main thread
                // Displays the name of currently executing Thread
         System.out.println("Currently Executing Thread :"+
Thread.currentThread().getName());
         System.out.println("Main thread priority :"+ Thread.currentThread().getPriority());
        // Main thread priority is set to 10
        Thread.currentThread().setPriority(10);
        System.out.println("Main thread priority :"+ Thread.currentThread().getPriority());
   }
}
```

Output

```
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Currently Executing Thread : main
Main thread priority : 5
Main thread priority : 10
```

Note:

- Thread with highest priority will get execution chance prior to other threads.
- Suppose there are 3 threads t1, t2 and t3 with priorities 4, 6 and 1.
- So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.
- Default priority for main thread is always 5, it can be changed later.
- Default priority for all other threads depends on the priority of parent thread.

Example 2

```
class X implements Runnable
{
public void run()
{
System.out.println("Thread X started");
```

```
for(int i = 1; i<=4; i++)
{
 System.out.println("Thread X: " +i);
 }
System.out.println("Exit from X");
 }
}
class Y implements Runnable
{
public void run()
{
 System.out.println("Thread Y started");
 for(int j = 0; j <= 4; j++)
 {
  System.out.println("Thread Y: " +j);
 }
 System.out.println("Exit from Y");
}
}
class Z implements Runnable
{
public void run()
{
 System.out.println("Thread Z started");
 for(int k = 0; k <= 4; k++)
 {
  System.out.println("Thread Z: " +k);
 }
 System.out.println("Exit from Z");
 }
}
class ThreadPriority
 {
public static void main(String[] args)
{
 X x = new X();
 Y y = new Y();
 Z z = new Z();

Thread t1 = new Thread(x);
Thread t2 = new Thread(y);
Thread t3 = new Thread(z);

t1.setPriority(Thread.MAX_PRIORITY);
t2.setPriority(t2.getPriority() + 4);
t3.setPriority(Thread.MIN_PRIORITY);

t1.start();
t2.start();
t3.start();
 }
}
```

Output

```
Thread X started
Thread Z started
Thread Z: 0
Thread Z: 1
Thread Z: 2
Thread Y started
Thread Z: 3
Thread X: 1
Thread Z: 4
Thread Y: 0
Thread Y: 1
Exit from Z
Thread X: 2
Thread Y: 2
Thread Y: 3
Thread X: 3
Thread Y: 4
Exit from Y
Thread X: 4
```

Output:

```
Thread X started
Thread Z started
Thread X: 1
Thread Z: 0
Thread Y started
Thread Z: 1
Thread X: 2
Thread X: 3
Thread X: 4
Exit from X
Thread Z: 2
Thread Z: 3
Thread Y: 0
Thread Y: 1
Thread Y: 2
Thread Y: 3
Thread Y: 4
Exit from Y
Thread Z: 4
Exit from Z
```

Exit from X

Synchronization in Java

Synchronization in java is the capability to control the access of multiple threads to any shared resource. Java Synchronization is better option where we want to allow only one thread to access the shared resource.

When we declare a synchronized keyword in the header of a method, it is called **synchronized method in Java**. Using synchronized keyword, we can synchronize all the methods of any class.

The main purpose of synchronization is to avoid thread interference. At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

Real time Example of Synchronization in Java

1. Suppose a thread in a program is reading a record from a file while another thread is still writing the same file. In this situation, the program may produce undesirable output.
2. Let's take a scenario of railway reservation system where two passengers are trying to book seats from Dhanbad to Delhi in Rajdhani Express. Both passengers are trying to book tickets from different locations.

Now suppose that both passengers start their reservation process at 11 am and observe that only two seats are available. First passenger books two seats and simultaneously the second passenger books one seat.

Since the available number of seats are only two but booked seats are three. This problem happened due to asynchronous access to the railway reservation system. In this realtime scenario, both passengers can be considered as threads, and the reservation system can be considered as a single object, which is modified by these two threads asynchronously. This asynchronous problem is known as **race condition** in which multiple threads access the same object and modify the state of object inconsistently.

The solution to this problem can be solved by a synchronization mechanism in which when one thread is accessing the state of object, other thread will wait to access the same object at a time until their come turn.

Types of Synchronization

- There are two types of synchronization
    - Process Synchronization :- The simultaneous execution of multiple threads or processes to reach a state such that they commit to a certain sequence of actions.
    - Thread Synchronization:- At times when more than one thread tries to access a shared resource, you need to ensure that resource will be used by only one thread at a time.

Thread Synchronization

- There are two types of thread synchronization mutual exclusive and inter-thread communication.
- Mutual Exclusive
    - Synchronized method.
    - Synchronized block.
    - static synchronization.
- Cooperation (Inter-thread communication in java)

Concept of Lock in Java

Synchronization is built around an internal entity known as the **lock** or **monitor**. Each and every object has a lock associated with it. So a thread that needs consistent access to an object's fields needs to acquire the object's lock before accessing them, and then release the lock when the work is done.

When a method is declared as synchronized, JVM creates a monitor (lock). To enter the monitor, the synchronized method is called. The thread that calls synchronized method first, acquires object lock. If the object lock is not available, calling thread is blocked and it has to wait until the lock becomes available. As long as thread holds lock (monitor), no other thread can enter the synchronized method and will have to wait for the current thread to release the lock on the same object. Look at the below figure to understand better.

Once a thread completes the execution of code inside the synchronized method, it releases object lock and allows other thread waiting for this lock to proceed. That is once a thread completes its work using synchronized method, it will hand over to the next thread that is ready to use the same resource.

A lock has two operations: acquire and release. The process of acquiring and releasing object lock (monitor) of an object is handled by Java runtime system (JVM).

Rules for synchronizing shared resources in Java

In the Java program, there must mainly three rules to be followed when synchronizing share resources. The rules are as follows:

1. A thread must get an object lock associated with it before using a shared resource. If a thread has an object lock of the shared resource, Java runtime system (JVM) will not allow another thread to access the shared resource.

If a thread is trying to access the shared resource at the same time, it is blocked by JVM and has to wait until the resource is available.

2. Only methods or blocks can be synchronized. Variables or classes cannot be synchronized.

3. Only one lock is associated with each object or shared resource.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
class Table
{
        void printTable(int n)
        {//method not synchronized
                for(int i=1;i<=5;i++)
                {
                        System.out.println(n*i);
                try
                {
                        Thread.sleep(400);
                }
                catch(Exception e){System.out.println(e);}
                }
        }
}

class MyThread1 extends Thread
{
        Table t;
        MyThread1(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(5);
        }
 }
class MyThread2 extends Thread
{
        Table t;
        MyThread2(Table t)
```

OUTPUT:

```
Called First thread
Called Second thread
5
100
200
10
300
15
20
400
25
500
```

```
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(100);
        }
}

class TestSynch1
{
        public static void main(String args[])
        {
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                System.out.println("Called First thread");
                t1.start();
                System.out.println("Called Second thread");
                t2.start();
        }
}
```

synchronized Keyword

Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

The synchronized keyword can be used to mark four different types of blocks:

1. methods synchronization
2. Synchronized block.
3. static synchronization

These blocks are synchronized on different objects.

1. Java synchronized method

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Synchronized methods enables a simple strategy for preventing the thread interference and memory consistency errors. If an Object is visible to more than one threads, all reads or writes to that Object's fields are done through the synchronized method.

It is not possible for two invocations for synchronized methods to interleave. If one thread is executing the synchronized method, all others thread that invoke synchronized method on the same Object will have to wait until first thread is done with the Object.
The syntax is,

```
synchronized return-type meth-name( )
{
        ..................
}
```

When we declare a method synchronized, java creates a monitor (lock) and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of the code. A monitor is like a key and the thread that holds the key can only open the lock.

//Example of java synchronized method

```
class Table
{
        synchronized void printTable(int n) //synchronized method
```

```
          {
                    for(int i=1;i<=5;i++)
                    {
                              System.out.println(n*i);
                              try
                              {
                                        Thread.sleep(400);
                              }
                              catch(Exception e){System.out.println(e);}
                    }
          }
}

class MyThread1 extends Thread
{
          Table t;
          MyThread1(Table t)
          {
                    this.t=t;
          }
          public void run()
          {
                    t.printTable(5);
          }
 }
class MyThread2 extends Thread
{
          Table t;
          MyThread2(Table t)
          {
                    this.t=t;
          }
          public void run()
          {
                    t.printTable(100);
          }
}

class TestSynch2
{
          public static void main(String args[])
          {
```

OUTPUT

```
Called First thread

Called second thread

5

10

15

20

25

100

200

300

400

500
```

```
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                System.out.println("Called First thread");
                t1.start();
                System.out.println("Called Second thread");
                t2.start();
}
}
```

2. Synchronized Block in Java

If want to synchronize access to an object of a class or only a part of a method to be synchronized then we can use synchronized block for it. It is capable to make any part of the object and method synchronized. Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

```
        synchronized (object reference expression)
        {
                //code block
        }
```

Example of synchronized block

```
class Table
```

```
{
           void printTable(int n)
          {
           synchronized(this)//synchronized block
          {
                     for(int i=1;i<=5;i++)
                     {
                               System.out.println(n*i);
                              try
                              {
                                        Thread.sleep(400);
                               }
                              catch(Exception e){System.out.println(e);}
                     }
          }
          }//end of the method
}

class MyThread1 extends Thread
{
          Table t;
          MyThread1(Table t)
          {
                    this.t=t;
          }
          public void run()
          {
                    t.printTable(5);
          }

}
class MyThread2 extends Thread
{
          Table t;
          MyThread2(Table t)
          {
                    this.t=t;
          }
          public void run()
          {
                    t.printTable(100);
          }
}

class TestSynchBlock1
{
        public static void main(String args[])
        {
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
        }
}
```

Difference between synchronized keyword and synchronized block

- When we use synchronized keyword with a method, it acquires a lock in the object for the whole method.
- It means that no other thread can use any synchronized method until the current thread, which has invoked it's synchronized method, has finished its execution.
- synchronized block acquires a lock in the object only between parentheses after the synchronized keyword.
- This means that no other thread can acquire a lock on the locked object until the synchronized block exits.
- But other threads can access the rest of the code of the method.

A synchronized method in Java is very slow and can degrade performance. So we must use synchronization keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only.

3. Static Synchronization

- Static synchronization is achieved by static synchronized methods.

- ☐ Static synchronized method locked on class and non-static synchronized method locked on current object.
- ☐ If static synchronized method is called a class level lock is acquired and then if an object is tries to access non-static synchronized method at the same time it will not be accessible because class level lock is already acquired.

Example of static synchronized method

```
class PrintTable
{
	public synchronized static void Table(int n)
	{
		System.out.println("Table of " + n);
		 for(int i=1;i<=10;i++)
		{
			System.out.println(n*i);
			try
			{
				Thread.sleep(500);
			 }
		catch(Exception e)
		{	System.out.println(e);  }
		}
	  }
}

class MyThread1 extends Thread
{
	public void run()
	{
		PrintTable.Table(2);
	}
}

class MyThread2 extends Thread
{
	public void run()
	{
		PrintTable.Table(5);
```

OUTPUT:

```
Table of 2
2
4
6
8
10
12
14
16
18
20
```

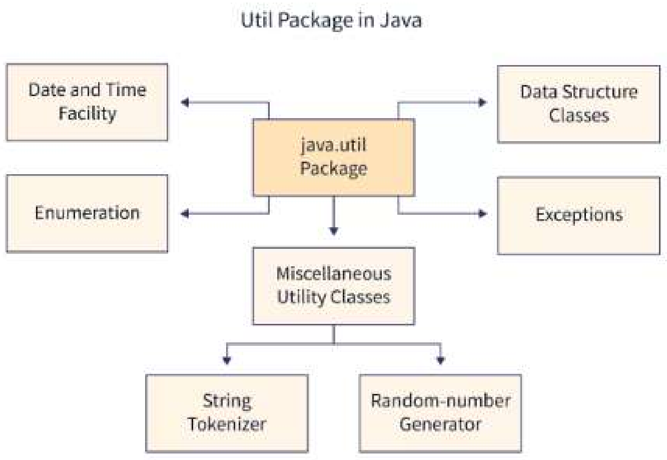
```
Table of 5
5
10
15
20
25
30
35
40
45
50
```

```
        }
}

public class StaticSync1
{
         public static void main(String args[])
        {
        //creating threads.
           MyThread1 t1=new MyThread1();
           MyThread2 t2=new MyThread2();
           //start threads.
           t1.start();
           t2.start();
        }
}
```

java.util Package in Java

The java.util package in Java consists of several components that provide basic necessities to the programmer. These components include:



Data Structure Classes: The java.util package contains several pre-written data structures like Dictionary, Stack, LinkedList, etc. that can be used directly in the program using the import statements.

Date and Time Facility: The java.util package provides several date and time-related classes that can be used to create and manipulate date and time values in a program.

Enumeration: It provides a special interface known as enumeration (enum for short) that can be used to iterate through a set of values.

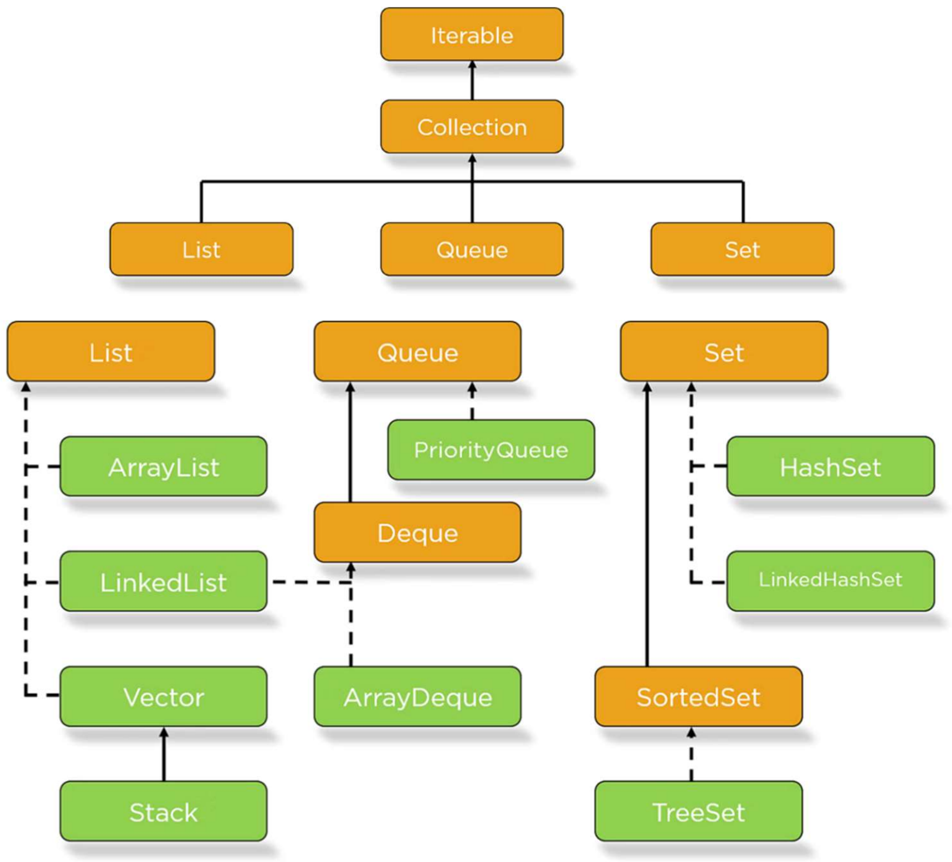
Exceptions: It provides classes to handle commonly occurring exceptions in a Java program.

Miscellaneous Utility Classes: The java.util package contains essential utilities such as the string tokenizer and random-number generator.

Java Collections Framework

The Java collections framework provides a set of interfaces and classes to implement various data structures and algorithms. Java Collection Framework offers the capability to Java Collection to represent a group of elements in classes and Interfaces. Java Collection Framework enables the user to perform various data manipulation operations like storing data, searching, sorting, insertion, deletion, and updating of data on the group of elements.

Java Collection Framework Hierarchy



- ☐ The Collections Framework in Java follows a certain hierarchy. All the classes and interfaces in the Collections Framework come under the java.util package.
- ☐ The above figure illustrates the hierarchy of the Collections Framework. It consists of four core interfaces such as Collection, List, Set, Queue, and various classes which get implemented through them.

Java Collection Interface

- The Collection interface is the root interface of the collections framework hierarchy.
- Sub interfaces of the Collection Interface
    - List Interface- The List interface is an ordered collection that allows us to add and remove elements like an array.
    - Set Interface- The Set interface allows us to store elements in different sets similar to the set in mathematics. It cannot have duplicate elements.
    - Queue Interface- The Queue interface is used when we want to store and access elements in First In, First Out manner.
    - Deque Interface- A Deque interface is inherited from the Java Collections Interface. The term DE-Que stands for Double-Ended Queue. The Deque supports insertion and deletion operations on both sides of the Queue.
    - Map Interface- The Map interface is inherited from Java Collection Interface. The Map cannot store duplicate elements. A Map stores data using the key-value pair format. The manipulation operations are taken care of through accessing the key-value pairs.
    - SortedSet Interface- Sorted set interface maintains mapping in ascending order. They are used for naturally ordered collection.
    - SortedMap Interface- A Sorted map interface maintains the mappings of elements in ascending critical order. SortedMap is the Map analog of SortedSet.

Methods of Collection

- The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its subinterfaces.
- add() - inserts the specified element to the collection
- size() - returns the size of the collection
- remove() - removes the specified element from the collection
- get()- helps to randomly access elements from lists
- iterator() - returns an iterator to access elements of the collection
- addAll() - adds all the elements of a specified collection to the collection
- removeAll() - removes all the elements of the specified collection from the collection

- ☐ clear() - removes all the elements of the collection

Java List

- ☐ In Java, the List interface is an ordered collection that allows us to store and access elements sequentially. It extends the Collection interface.
- ☐ Classes that Implement List
- ☐ Since List is an interface, we cannot create objects from it.
- ☐ In order to use the functionalities of the List interface, we can use these classes:
  - ArrayList
  - LinkedList
  - Vector
  - Stack
- ☐ The List interface includes all the methods of the Collection interface. Its because Collection is a super interface of List.
- ☐ In Java, we must import java.util.List package in order to use List.
- ☐ Different ways for creating object for List interface are
  - // ArrayList implementation of List

List<data type> list1 = new ArrayList<>();

eg: List<Integer> list1 = new ArrayList<>();

- // LinkedList implementation of List

List<data type> list2 = new LinkedList<>();

eg: List<String> list2 = new LinkedList<>();

Here, we have created objects list1 and list2 of classes ArrayList and LinkedList. These objects can use the functionalities of the List interface.

- List <data-type> list3 = new Vector<>();

eg: List<Integer> list3 = new Vector<>();

- List <data-type> list4 = new Stack();

eg: List <String> list4 = new Stack();

Java ArrayList

- ☐ In Java, we use the ArrayList class to implement the functionality of resizable-arrays.
- ☐ Before using ArrayList, we need to import the java.util.ArrayList package first. Here is how we can create arraylists in Java:
- ☐ ArrayList<Type> a= new ArrayList<>();
- ☐ Here, Type indicates the type of an Arraylist.
- ☐ For example,
  - ◘ // create Integer type Arraylist

  ArrayList<Integer> a = new ArrayList<>();
  - ◘ // create String type Arraylist

  ArrayList<String> s = new ArrayList<>();

Arraylist Methods in Java

- ☐ add(): It defines that an element Adds to the end of the Arraylist.
- ☐ add(index, element): Inserts an element at a specific index in the Arraylist.
- ☐ remove(index): Removes the element at a specific index in the Arraylist.
- ☐ get(index): Returns the element at a specific index in the ArrayList.
- ☐ size(): It defines the number of elements in the Arraylist that is returned using the size operator.
- ☐ clear(): Removes all elements from the Arraylist.
- ☐ indexOf(element): It defines the index of the first occurrence of the specified element in the Arraylist that has been returned.
- ☐ lastIndexOf(element): It defines the index of the last occurrence of the specified element in the ArrayList that has returned.
- ☐ sort(): Sorts the Arraylist in ascending order.
- ☐ Collections. sort() method

Example 1:

```
import java.util.*;

class arrayExample
```

```
{
   public static void main (String[] args) throws java.lang.Exception
   {
      // your code goes here
       ArrayList<String> names = new ArrayList<>();
      names.add("Sharma");
      names.add("Gourav");
      names.add("Deepthi");
      System.out.println(names);
      String n1 = names.get(1);
      System.out.println(n1);
      int index = names.indexOf("Gourav");
      System.out.println(index);
      Collections.sort(names);
      System.out.println(names);
      names.remove(1);
      System.out.println(names);
   }
}
```

Output

```
[Sharma, Gourav, Deepthi]
Gourav
1
[Deepthi, Gourav, Sharma]
[Deepthi, Sharma]
```

Example 2: Implementing the ArrayList Class using List interface

```
import java.util.List;
import java.util.ArrayList;
class Main {
  public static void main(String[] args) {
    // Creating list using the ArrayList class
    List<Integer> numbers = new ArrayList<>();
    // Add elements to the list
    numbers.add(10);
    numbers.add(20);
    numbers.add(30);
    System.out.println("List: " + numbers);
    // Access element from the list
    int number = numbers.get(2);
    System.out.println("Accessed Element: " + number);
    // Remove element from the list
    int removedNumber = numbers.remove(1);
    System.out.println("Removed Element: " + removedNumber);
  }
}
```

Output

```
List: [10, 20, 30]
Accessed Element: 30
Removed Element: 20
```

Java LinkedList

- The LinkedList class of the Java collections framework provides the functionality of the linked list data structure (doubly linkedlist).

- ☐ Here is how we can create linked lists in Java:
- ☐ LinkedList<Type> linkedList = new LinkedList<>();
- ☐ Here, Type indicates the type of a linked list. For example,
    - ◘ // create Integer type linked list

      LinkedList<Integer> linkedList = new LinkedList<>();
    - ◘ // create String type linked list

      LinkedList<String> linkedList = new LinkedList<>();

Example 1:

```
import java.util.LinkedList;
class Main {
  public static void main(String[] args){
    // create linkedlist
    LinkedList<String> animals = new LinkedList<>();
    // add() method without the index parameter
    animals.add("Dog");
    animals.add("Cat");
    animals.add("Cow");
    System.out.println("LinkedList: " + animals);
    // add() method with the index parameter
    animals.add(1, "Horse");
    System.out.println("Updated LinkedList: " + animals);
    // get the element from the linked list
    String str = animals.get(1);
    System.out.println("Element at index 1: " + str);
     // change elements at index 3
    animals.set(3, "Elephent");
    System.out.println("Updated LinkedList: " + animals);
    // remove elements from index 1
    String str1 = animals.remove(1);
    System.out.println("Removed Element: " + str1);
    System.out.println("Updated LinkedList: " + animals);
  }
}
```

Output

```
LinkedList: [Dog, Cat, Cow]
Updated LinkedList: [Dog, Horse, Cat, Cow]
Element at index 1: Horse
Updated LinkedList: [Dog, Horse, Cat, Elephent]
Removed Element: Horse
Updated LinkedList: [Dog, Cat, Elephent]
```

Example 2: Implementing the LinkedList Class using List interface

```
import java.util.List;
import java.util.LinkedList;
class Main {
    public static void main(String[] args) {
        // Creating list using the LinkedList class
        List<Integer> numbers = new LinkedList<>();
        // Add elements to the list
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);
        System.out.println("List: " + numbers);
        // Access element from the list
        int n = numbers.get(2);
        System.out.println("Accessed Element: " + n);
// Using the indexOf() method
        int index = numbers.indexOf(20);
        System.out.println("Position of 20 is " + index);
        // Remove element from the list
        int removedNumber = numbers.remove(1);
        System.out.println("Removed Element: " + removedNumber);
        System.out.println("LinkedList: " + numbers);
    }
}
```

Output

```
List: [10, 20, 30]
Accessed Element: 30
Position of 20 is 1
Removed Element: 20
LinkedList: [10, 30]
```

Java Stack Class

- ☐ The Java collections framework has a class named Stack that provides the functionality of the stack data structure.
- ☐ In order to create a stack, we must import the java.util.Stack package first. Once we import the package, here is how we can create a stack in Java.
- ☐ Stack<Type> stacks = new Stack<>();
- ☐ Here, Type indicates the stack's type. For example,
    - ◘ // Create Integer type stack

```
        Stack<Integer> stacks = new Stack<>();
```

- // Create String type stack

```
        Stack<String> stacks = new Stack<>();
```

Example 1: Implement stack

```
   import java.util.*;
   public class ScalerTopics{
   public static void main(String args[]){
   //creating a Stack
   Stack<Integer> s= new Stack<>();
      //displaying the initial size
   System.out.println("Size at the beginning "+s.size());
      //push elements
   s.push(99);
   s.push(28);
   s.push(17);
   s.push(74);
   s.push(1);
      //displaying the Stack
   System.out.println("New Stack" + s);
      //displaying the size
   System.out.println("Size after addition "+s.size());
     //pop the element and display it
   System.out.println("Popped element " + s.pop());
      //display the new Stack
   System.out.println("New Stack after popping"+ s);
      //display the new size
```

```
System.out.println("Size after removal "+s.size());
   //peek method to find the top-most element and display it
System.out.println("Top-most element " + s.peek());
   //the size remains the same as peek does not remove the element
System.out.println("Size after Peek() "+s.size());
}
}
```

Output

```
Size at the beginning 0
New Stack[99, 28, 17, 74, 1]
Size after addition 5
Popped element 1
New Stack after popping[99, 28, 17, 74]
Size after removal 4
Top-most element 74
Size after Peek() 4
```

Queue Interface

- The Queue interface of the Java collections framework provides the functionality of the queue data structure. It extends the Collection interface.
- Since the Queue is an interface, we cannot provide the direct implementation of it.
- In order to use the functionalities of Queue, we need to use classes that implement it:
    - ArrayDeque
    - LinkedList
    - PriorityQueue
- Interfaces that extend Queue
    - The Queue interface is also extended by various subinterfaces:
        - Deque

  - ☐ BlockingQueue
  - ☐ BlockingDeque

- ☐ In Java, we must import java.util.Queue package in order to use Queue.
- ☐ // LinkedList implementation of Queue

  Queue<String> animal1 = new LinkedList<>();
- ☐ // Array implementation of Queue

  Queue<String> animal2 = new ArrayDeque<>();
- ☐ // Priority Queue implementation of Queue

  Queue<String> animal3 = new PriorityQueue<>();

  Here, we have created objects animal1, animal2 and animal3 of classes LinkedList, ArrayDeque and PriorityQueue respectively. These objects can use the functionalities of the Queue interface
- ☐ **Methods of Queue**
- ☐ The Queue interface includes all the methods of the Collection interface. It is because Collection is the super interface of Queue.
- ☐ Some of the commonly used methods of the Queue interface are:
- ☐ add() - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.
- ☐ offer() - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.
- ☐ element() - Returns the head of the queue. Throws an exception if the queue is empty.
- ☐ peek() - Returns the head of the queue. Returns null if the queue is empty.
- ☐ remove() - Returns and removes the head of the queue. Throws an exception if the queue is empty.
- ☐ poll() - Returns and removes the head of the queue. Returns null if the queue is empty.

Example 1: Implementing the LinkedList using Queue

```
import java.util.Queue;
import java.util.LinkedList;
class Main {
   public static void main(String[] args) {
      // Creating Queue using the LinkedList class
      Queue<Integer> numbers = new LinkedList<>();

      // offer elements to the Queue
      numbers.offer(1);
      numbers.offer(2);
      numbers.offer(3);
      System.out.println("Queue: " + numbers);

      // Access elements of the Queue
      int accessedNumber = numbers.peek();
      System.out.println("Accessed Element: " + accessedNumber);
      // Remove elements from the Queue
      int removedNumber = numbers.poll();
      System.out.println("Removed Element: " + removedNumber);
      System.out.println("Updated Queue: " + numbers);
   }
}
```

Output

```
Queue: [10, 20, 30]
Accessed Element: 10
Removed Element: 10
Updated Queue: [20, 30]
```

Example 2: Implementing the PriorityQueue using Queue Class

```
import java.util.Queue;
import java.util.PriorityQueue;
class Main {
   public static void main(String[] args) {
      // Creating Queue using the PriorityQueue class
      Queue<Integer> numbers = new PriorityQueue<>();
      // offer elements to the Queue
      numbers.offer(5);
      numbers.offer(1);
      numbers.offer(2);
      System.out.println("Queue: " + numbers);
      // Access elements of the Queue
      int accessedNumber = numbers.peek();
      System.out.println("Accessed Element: " + accessedNumber);
      // Remove elements from the Queue
      int removedNumber = numbers.poll();
      System.out.println("Removed Element: " + removedNumber);
      System.out.println("Updated Queue: " + numbers);
   }
}
```